Report

Sameer Gopali

March 2023

1 Implementation Details

This report covers the implementation details for Simpletorch library which follows similar API provided by the Pytorch. The library supports Linear layer, ReLU and Sigmoid Activation. The crossentropy loss, hinge loss, and mean squared loss is supported. For intialization, zero weight initializaiton and normal weight intialization is supported.
The library has simple implementation of reverse mode automatic differentiation. In the forward pass, it will create a computation graph. And durinig the backward pass, this graph is traveresed recursively to calculate gradient using chain multiplication.

1.1 Tensor

Tensor class represents a tensor data and it holds a numpy array. The numpy array is stored in the data attribute of the Tensor object. The Tensor class also has an autogradMeta object, which is a node in the computation graph. If tensor object requries gradient, then requires_grad attribute is set to True. If the tensor is not the result from any computation, then it will have a accumulator object, which is a special type of autogradMeta object that aggregrates all the gradient in computation graph. If the tensor object is result of any computation, then it's autogradMeta object will hold gradient function of the computing function.

1.2 AutoGradMeta

This class represents a node in computation graph. It holds the gradient function which computes gradient with respect to inputs in grad_fn attribute. These inputs will also have autogradMeta object which are stored in next_fn list. During backpropagation, the gradient function in the next_fn will be called to perform chain multiplication.

1.3 Function

This is a python module containing classes that define different mathematical functions, including forward and backward methods, used in building a compu-

tational graph for automatic differentiation. The Function class is the base class from which other classes inherit. It contains methods for forward and backward operations. The following functions are defined in library:

- The AddFunction class inherits from the Function class and defines the forward and backward operations for addition
- The ConstantMultiplication class inherits from the Function class and defines the forward and backward operations for a constant multiplication operation.
- The LinearFunction class inherits from the Function class and defines the forward and backward operations for a linear transformation.
- The sigmoid class inherits from the Function class and defines the forward and backward operations for the sigmoid activation function.
- The relu class inherits from the Function class and defines the forward and backward operations for the ReLU activation function.
- The softmax_crossentropy_logits class inherits from the Function class and defines the forward and backward operations for the softmax activation function with cross-entropy loss.
- The mean_squared_error class inherits from the Function class and defines the forward and backward operations for the mean squared error loss function.
- The hinge_loss class implements the function to calculate forward and backward operations.

.

1.4 Forward Pass

In forward mode, when any function defined in the library is called, it will compute the forward result of the forward function and create a Tensor. Then it will set the autogradMeta with it's backward function. It will also add the input's autogradMeta object into the next_fn list of the output tensor.

1.5 Backward Pass

During backpropagation, first the graph is traversed to recursively traversed to calculate the dependency number for each node. When we call the backward on the root node, it will call its gradient function and add the resultant gradient value in grad_buffer of child node in next_fn list. It will also decrement the dependency number for all the children. If the dependency number is zero, it will traverse the children recursively. Final gradient will be accumulated in the accumulator object.

1.6 CrossEntropyWithLogits

For numerical stability we first calcuate the logsoftmax using the logits. It is calcualted as follows:

```
c = input.data.max(axis=1).reshape(-1,1)
logsumexp = np.log(np.exp(input.data - c).sum(axis=1)).reshape(-1,1)
return input.data - c - logsumexp
```

First it computes the maximum value of each row of the input Tensor and subtracts it from the input Tensor, Then it exponentiates each element of the resulting Tensor, and finally computes the logarithm of the sum of the exponentiated values of the corresponding row. After this we subtract with the input value to get log softmax value. Finally, the crossentropy loss will be calculated by multiplying with the actual value.

1.7 HingeLoss

The hinge loss is calculated as follows:

$$\sum_{i \neq j} max\{1 + \hat{y}_i - \hat{y}_j, 0\}$$

```
# get prediction value for true class
true_pred = (input.data * actual.data).sum(axis=-1).reshape(-1,1)
# get predictions value for other classes
other_pred = (1. - actual.data) * input.data
# calculate margins for all predictions by broadcasting
self.margin = other_pred  - true_pred +1
#  max(margin,0)
out = np.where(self.margin>0, self.margin,0)
# We don't sum for true class, so remove it
out = np.where(actual.data==1, 0, out)
# sum over values
out = out.sum(axis=1)
# Reduce with mean
out = out.mean()
```

1.8 Linear layer

In the forward pass, Linear layer takes in NxF matrix(N=no. of samples and F=no. of features) as input and applies linear transformation. The parameters for this layer are weight and bias matrix. In the backward pass gradient for input, weigh and bias is calculated as follows:

```
# grad_ouptut is the gradient from the next layer flowing backward.
grad_input= np.matmul(grad_output,self.weight.data)
grad_weight= np.matmul(grad_output.T,self.input.data)
grad_bias=  grad_output.sum(0)
```

2 References

[1] https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/
[2] https://pytorch.org/blog/how-computational-graphs-are-executed-in-pytorch/